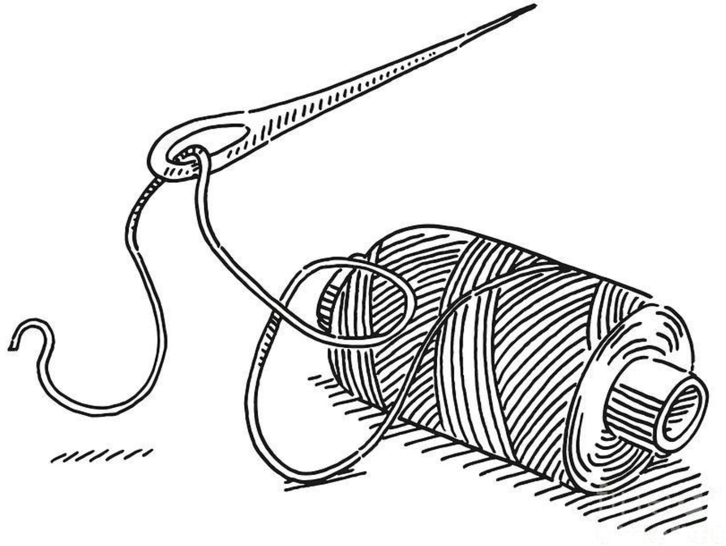
Threads Cont. Locks & Concurrency Benefits

Computer Operating Systems, Fall 2025



Instructors: Joel Ramirez

Head TAs: Maya Huizar, Akash Kaukuntla, Vedansh Goenka, Joy Liu

TAs:

Eric Zou	Joseph Dattilo	Aniket Ghorpade	Shriya Sane
Zihao Zhou	Eric Lee	Shruti Agarwal	Yemisi Jones
Connor Cummings	Shreya Mukunthan	Alexander Mehta	Raymond Feng
Bo Sun	Steven Chang	Rania Souissi	Rashi Agrawal
Sana Manesh			

Administrivia

- Penn-shell is due Monday, October 6th
  - If you have any questions, please stop by office hours! We are here to help.
  - Latest time to turn it in is Friday, October 10th during fall break…don't do this to yourself.

- Midterm will be on Thursday, October 16 from 5:15PM – 6:45PM
  - Locations: Towne 100 & Wu and Chen in Levine
  - Towne 100
    - Last time starts with: A – M
  - Wu and Chen
    - Last time starts with: N - Z
  - Practice exams and the official post will go out later today

Administrivia

- PennOS:
  - Specifications and team sign-up to be posted Friday (day after exam)
  - Done in groups of 4
  - Partner signup due by end of day on Monday, **10/20**
    - Those left unassigned will be randomly assigned the next morning (Tuesday the 21st)
  - Lecture dedicated to PennOS in class on Tuesday the 21st. ***Highly recommend you go.***

- ***No Check-in*** **10/14!**
  - ***Study for the exam…come to the exam review that Tuesday! ☺***

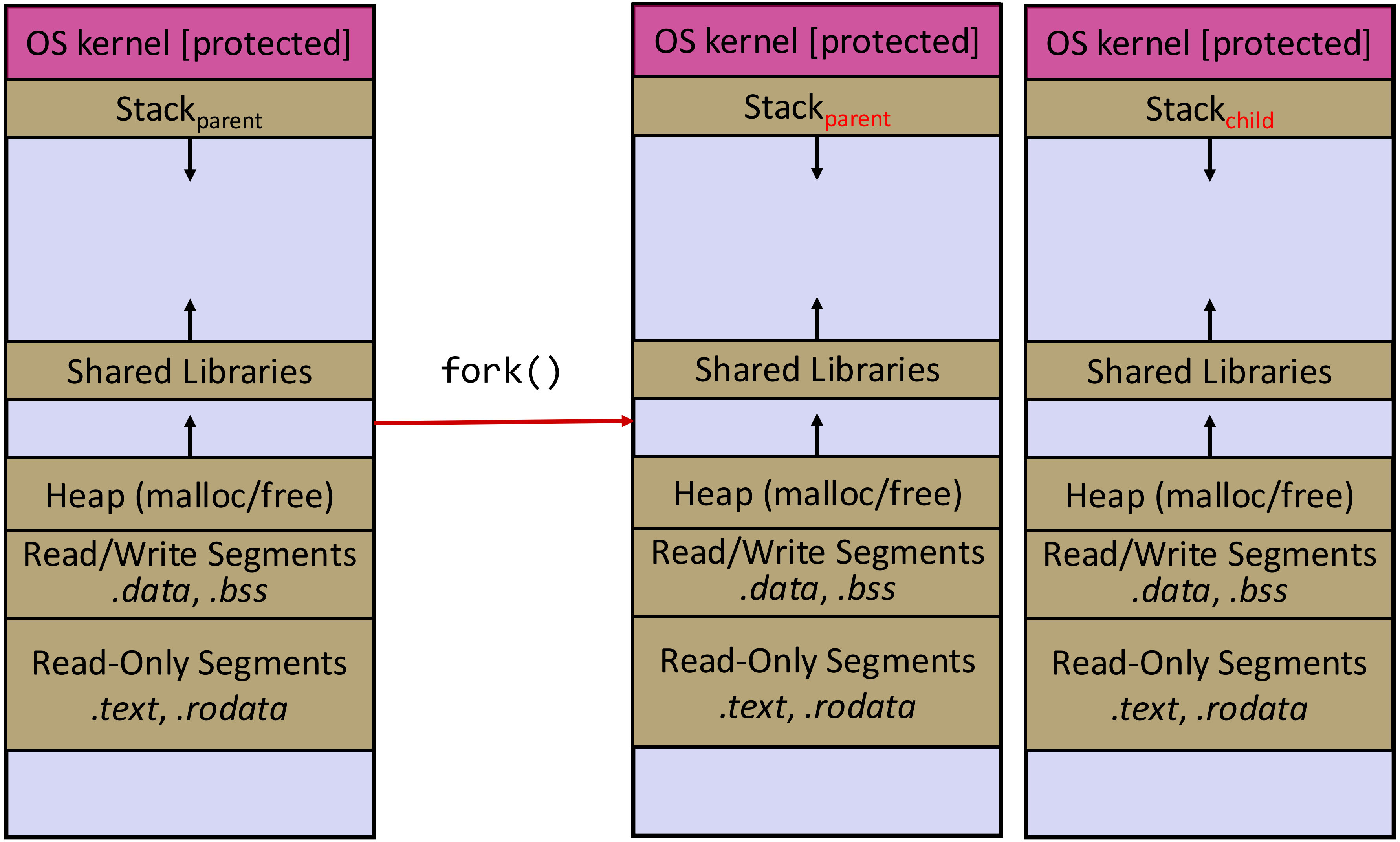
Lecture Outline

- **Threads Refresher**
- Synchronization Mechanisms
- Data Race vs Race Condition
- Is a mutex needed? (Peterson’s)
- It only gets worse

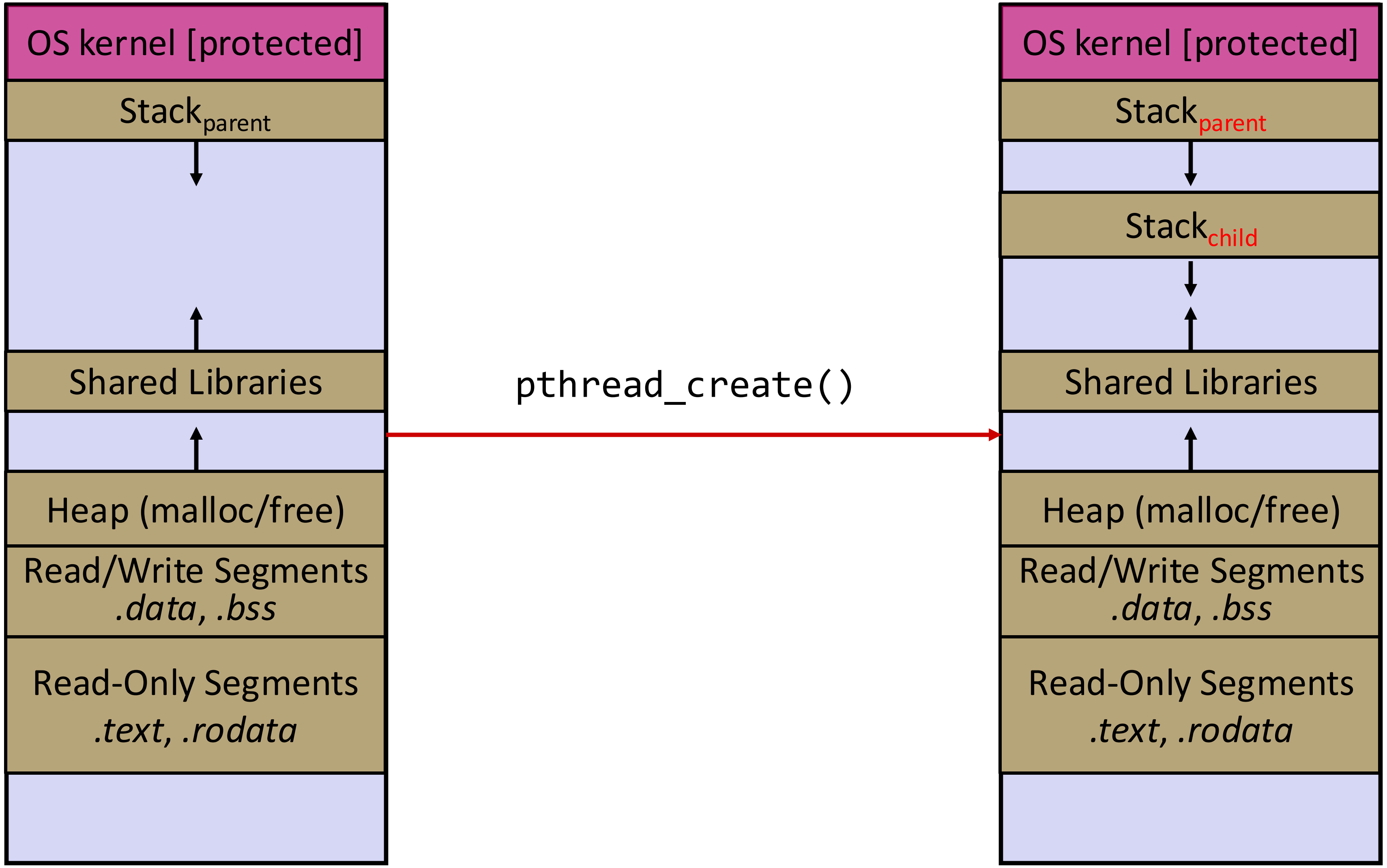
Threads vs. Processes

- In most modern OS's:
  - A Process has a unique: address space, OS resources, & security attributes
  - A Thread has a unique: stack, stack pointer, program counter, & registers
  - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

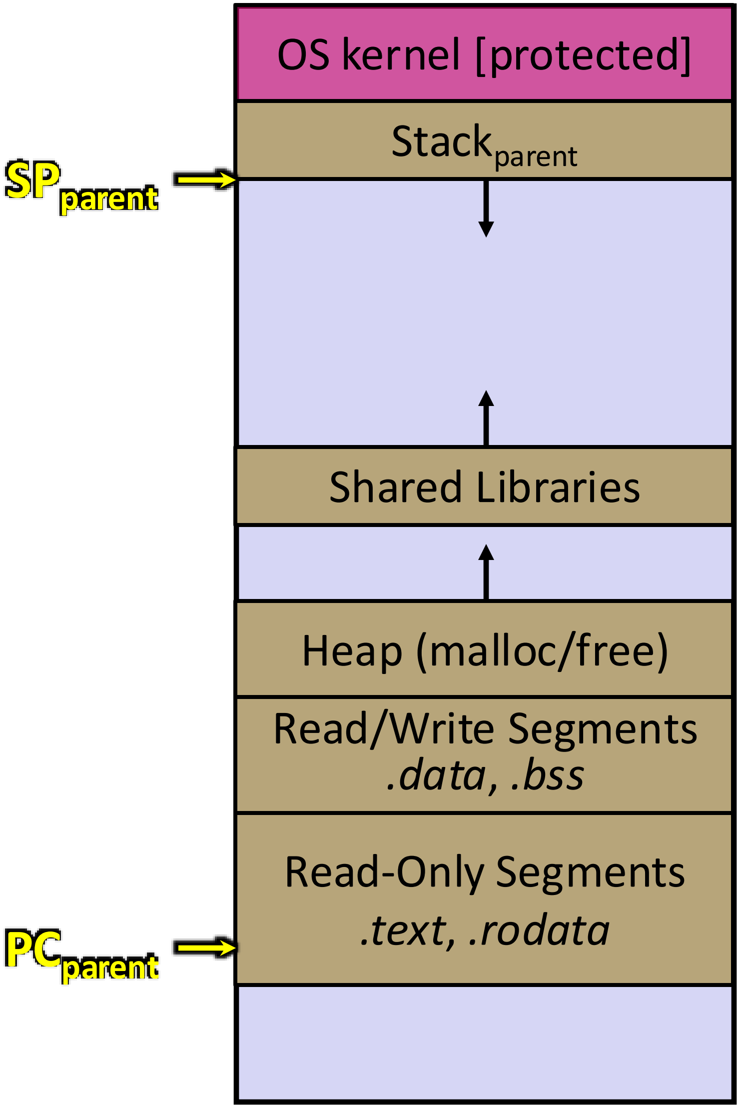
Threads vs. Processes



Threads vs. Processes

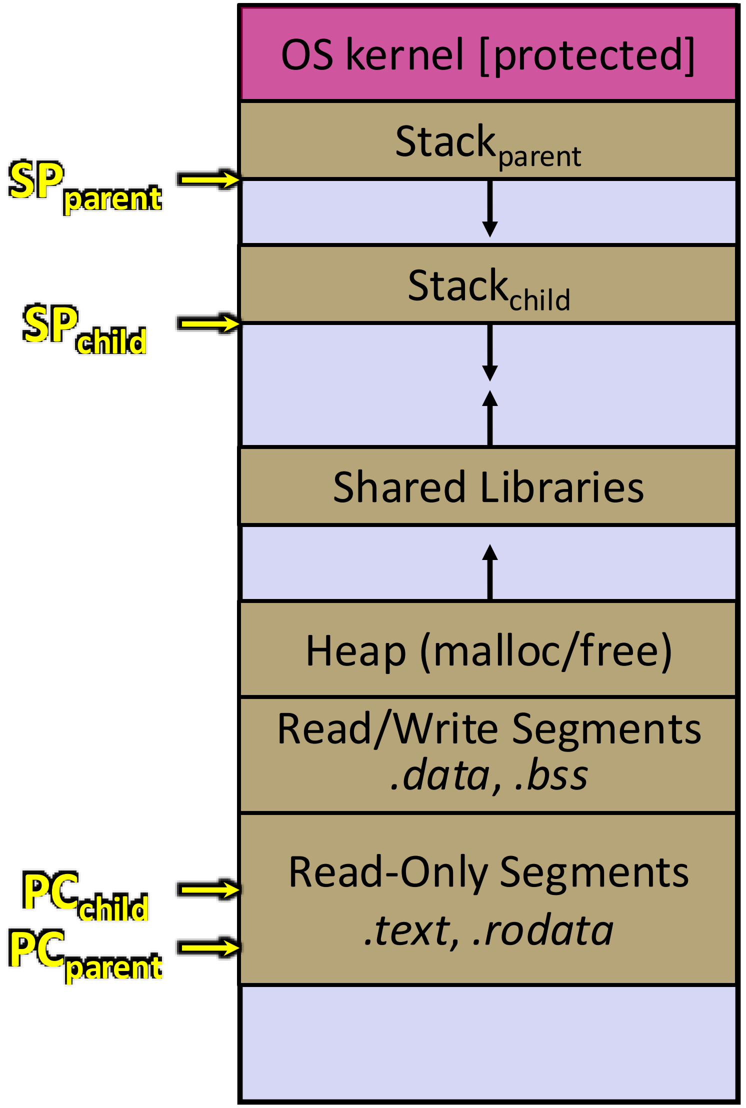


Single-Threaded Address Spaces



- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`

Multi-threaded Address Spaces



- After creating a thread
  - *Two* threads of execution running in the address space
    - Original thread (parent) and new thread (child)
    - New stack created for child thread
    - Child thread has its own *values* of the PC and SP
  - Both threads share the other segments (code, heap, globals)
    - They can cooperatively modify shared data

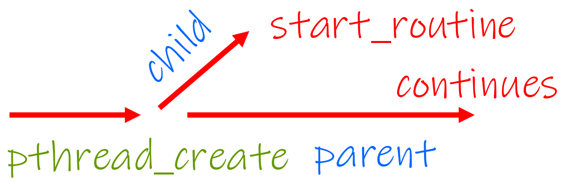
POSIX Threads (pthreads)

- The POSIX APIs for dealing with threads
  - Declared in `pthread.h`
    - Not part of the C/C++ language
  - To enable support for multithreading, must include `-pthread` flag when compiling and linking with `clang-15` command
    - `clang-15 -g -Wall -pthread -o main main.c`
  - Implemented in C
    - Must deal with C programming practices and style

Creating and Terminating Threads

```
int pthread_create(
        pthread_t* thread,
        const pthread_attr_t* attr,
        void* (*start)(void*),
        void* arg);
```

- Creates a new thread with attributes `*attr`
- Returns `0` on success and an error number on error (can check against error constants)`
- The new thread runs `start(arg)`



This uses our previous conception of child vs parent metaphor, but really, they are not treated this way. There is no hierarchy of threads. They are more like siblings.

pthread_create in reality

```
int pthread_create(
        pthread_t* thread,
        const pthread_attr_t* attr,
        void* (*start)(void*),
        void* arg);
```

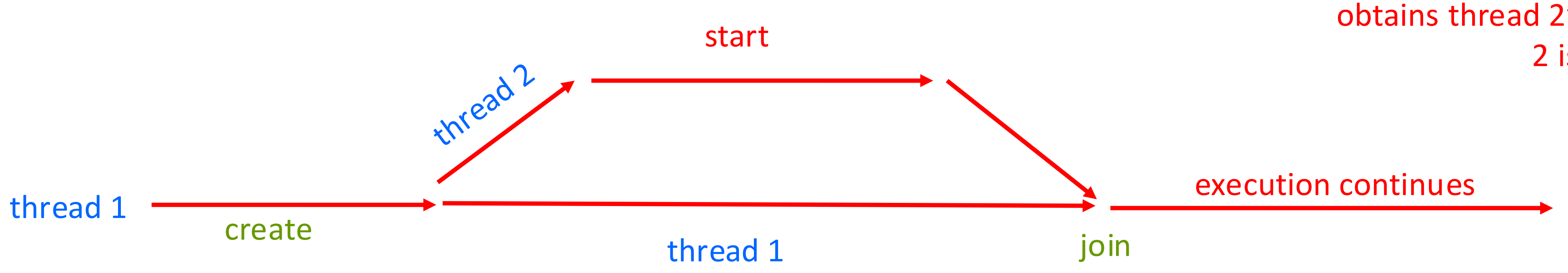
- `pthread_t* thread`
  - Output parameter: gives us a thread identifier
  - Varies from OS to OS, in Linux it is an unsigned long in others, a struct.
- `const pthread_attr_t* attr`
  - An struct detailing the attributes that the thread will take on. Null for default attributes.
- `void* (*start)(void*)`
  - Function that the newly created thread will commence executing from.
    - (i.e. `void *start(void *arg)`)
- `void* arg`: the argument that will be passed into start (you can do a lot with a ptr)

We Created a Thread, Now What?

```
int pthread_join(pthread_t thread, void** retval);
```

- The calling thread waits for the thread specified by `thread` to terminate, and as the name says, joins the two executions stream into one (hence, "join")
- You can think of it as the thread equivalent of `waitpid()`, although it really isn't.
- The exit status of the terminated thread is placed in `**retval`

thread 1 waits for thread 2 to exit, it obtains thread 2's return value, and thread 2 is cleaned up



No more, parent and child. JUST THREADS!

Any thread within the same process can join with any other thread.

The Pain Point: Shared Resources

- POSIX.1: Some resources are shared between threads and processes
- All Threads Share the Following
  - PID, Parent PID, PGID, Controlling Terminal, File Descriptors, Interval timers (sleep, alarm…)
  - nice value (niceness applies *per process*, not per thread according to POSIX)
  - ***global memory (data and heap segments)*** ← ***Our focus***
- The following attributes are distinct for each thread
  - Stack
  - thread ID (the *pthread_t* data type)
  - signal mask (pthread_sigmask(3))
  - the *errno* variable

Data Races

- Two memory accesses form a data race if different threads access the same location, ***at least one is a write, and the accesses can co-occur.***
  - The state of a program can vary depending on scheduling…
    - **Which thread ran first?**
    - **When did a thread get interrupted?**

Data Race Example

- If your fridge has no milk, then go out and buy some more
  - What could go wrong?

```
if (!milk) {
  buy(milk);
}
```

- If you live alone:

- If you live with a roommate:

Poll Everywhere

pause and think!

- Idea: leave a note!
  - Does this fix the problem
    - (with two threads, not roommates)

```
if (!note) {
  if (!milk) {
    leave(note);
    buy(milk);
    remove(note);
  }
}
```

A. **Yes, problem fixed**

B. **No, could end up with no milk**

C. **No, could still buy multiple milk**

D. **We’re lost…**

Poll Everywhere

pause and think!

- Idea: leave a note!
  - Does this fix the problem
    - (with two threads, not roommates)

```
if (!note) {
  if (!milk) {
    leave(note);
    buy(milk);
    remove(note);
  }
}
```

you	roommate
Check note	
	Check note
Check milk	
Leave note	
	Check milk
	Leave note
	Buy milk
Buy milk	

time

A. Yes, problem fixed

B. No, could end up with no milk

C. No, could still buy multiple milk

D. We're lost…

We can be interrupted between checking note and leaving note ☹

*There are other possible scenarios that result in multiple milks

Nothing has changed!

We are still reading and modifying a value; `note`.

Threads and Data Races

- Data races might interfere in painful, non-obvious ways, depending on the specifics of the data structure
- Example: two threads try to read from and write to the same shared memory location
  - Could get "correct" answer
  - Could accidentally read old value
  - One thread's work could get "lost"
- Example: two threads try to push an item onto the head of the linked list at the same time
  - Could get "correct" answer
  - Could get different ordering of items
  - Could break the data structure! ☠

This should remind you of the signal interruption!

We were essentially interrupting one stream of execution with another!

Remember this?

- What does this print?

Always prints 0, the global counter is not shared across processes, so the parent's global never changes

```c
#define NUM_PROCESSES 50
#define LOOP_NUM 100

int sum_total = 0;

void loop_incr() {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  printf("Process ID: %d with sum total of %d.\n", getpid(), sum_total);
}

int main(int argc, char** argv) {

  pid_t pids[NUM_PROCESSES];  // array of process ids

  // create processes to run loop_incr()
  for (int i = 0; i < NUM_PROCESSES; i++) {
    pids[i] = fork();
    if (pids[i] == 0) {    // child
      loop_incr();
      exit(EXIT_SUCCESS);
    }
  }
  // wait for all child processes to finish
  for (int i = 0; i < NUM_PROCESSES; i++) {
    waitpid(pids[i], NULL, 0);
  }

  printf("The ultimate sum total is %d\n", sum_total);
  return EXIT_SUCCESS;
}
```

Remember this?

- What does this print?

Usually 5000

```
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

void *loop_incr(void *arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  printf("Thread ID: %d with sum total of %d.\n", gettid(), sum_total);
  return NULL;
}


int main(int argc, char** argv) {
  pthread_t thds[NUM_THREADS];  // array of thread ids

  // create threads to run thread_main()
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_create(&thds[i], NULL, &loop_incr, NULL) != 0) {
      fprintf(stderr, "pthread_create failed\n");
    }
  }

  // wait for all non-main threads to finish
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(thds[i], NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
    }
  }

  printf("The ultimate sum total is %d\n", sum_total);
  return EXIT_SUCCESS;
}
```

Previous Demos:

- See `total.c` and `total_processes.c`
  - Threads share an address space, if one thread increments a global, it is seen by other threads
  - Processes have separate address spaces, incrementing a global in one process does not increment it for other processes

- NOTE: sharing data between threads is unsafe if done wrong (we are doing it wrong in this example); let’s expand on that now ☺

Increment Data Race

- What can be written in one line in c `sum_total++` is multiple assembly instructions in one. The increment looks something like this in assembly:

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

Equivalent pseudo →

```
a0  = *t0
a0  = a0 + 1
*t0 = a0
```

- What happens if we context switch to a different thread while executing these three instructions?

- **Reminder: Each thread has its own registers to work with. Each thread would have its own** `a0 & t0`
  - But, they would share the memory location stored within `t0`

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

`sum_total = 0`

Thread 0 `a0 = 0`

```
lw a0, 0(t0)
```

Thread 1

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 a0 = 0

```
lw a0, 0(t0)
```

Thread 1 a0 = 0

```
lw a0, 0(t0)
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 a0 = 0

```
lw a0, 0(t0)
```

Thread 1 a0 = 1

```
lw a0, 0(t0)
addi a0, a0, 1
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 a0 = 0

```
lw a0, 0(t0)
```

Thread 1 a0 = 1

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

`sum_total = 1`

Thread 0 `a0 = 1`

```
lw a0, 0(t0)



addi a0, a0, 1
```

Thread 1 `a0 = 1`

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 a0 = 1

```
lw a0, 0(t0)



addi a0, a0, 1
sw a0, 0(t0)
```

Thread 1 a0 = 1

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

- With this example, we could get 1 as an output instead of 2, even though we executed addi twice.

Poll Everywhere

pause and think

- What is the minimum value that could be printed by main thread?
  - Single Core
  - Concurrent Threading Only

- Important: all three exec on each loop

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

- Joel said 100 in lecture on Thursday, but this only applies in ***one*** specific scenario.

```c
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

void *loop_incr(void *arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  printf("Thread ID: %d with sum total of %d.\n", gettid(), sum_total);
  return NULL;
}


int main(int argc, char** argv) {
  pthread_t thds[NUM_THREADS];  // array of thread ids

  // create threads to run thread_main()
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_create(&thds[i], NULL, &loop_incr, NULL) != 0) {
      fprintf(stderr, "pthread_create failed\n");
    }
  }

  // wait for all non-main threads to finish
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(thds[i], NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
    }
  }

  printf("The ultimate sum total is %d\n", sum_total);
  return EXIT_SUCCESS;
}
```

Poll Everywhere

pause and think

- Most common mistake: ***100***.
- One of the scenarios:

Thread 1

```
lw a0, 0(t0) //loads 0



addi a0, a0, 1
sw a0, 0(t0) //stores 1
// 99 more times
```

100!

Threads 2 - 50

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

all finish before thread 1 stores for first time

Poll Everywhere

pause and think

- What is the minimum value that could be printed?
  - Single Core
  - Concurrent Threading Only

Thread 1

```
lw a0, 0(t0) //loads 0


addi a0, a0, 1
sw a0, 0(t0) //stores 1

//finishes executing
```

Thread 2

```
//does loop 99 X



lw a0, 0(t0)


addi a0, a0, 1
sw a0, 0(t0)
```

finish....

loads 1!

stores 2!

Threads 3 - 50

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

finish before thread 1 stores for first time

Poll Everywhere

pause and think

- True Minimum Value: True for any multithreaded program with any # of increments.

Thread 1

```
lw a0, 0(t0) //loads 0

addi a0, a0, 1
sw a0, 0(t0) //stores 1

//finishes executing
```

Thread 2

```
//does loop 99 X

lw a0, 0(t0)

addi a0, a0, 1
sw a0, 0(t0)
```

finish….

Threads 3 - 50

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

finish before thread 1 stores for first time

loads 1!

stores 2!

If we need to repeatedly load and store to a global: The minimum is 2.

Poll Everywhere

pause and think

- Things to consider:
  - When are values loaded? Every time? Before the loop? What if there are optimizations?
  - With optimizations, the minimum is no longer 2. ***It is 100.***

```
for 100 times:
   lw a0, 0(t0) //loads 0
   addi a0, a0, 1
   sw a0, 0(t0) //stores 1
```

-O1

```
lw a0, 0(t0) //loads global
addi a0, a0, 100
sw a0, 0(t0) //stores global + 100
```

Lecture Outline

- Threads Refresher
- Synchronization Mechanisms
- Data Race vs Race Condition
- Is a mutex needed? (Peterson's)
- It only gets worse

Synchronization

- Synchronization is the act of preventing two (or more) concurrently running threads from interfering with each other when operating on shared data
  - Need some mechanism to coordinate the threads
    - “Let me go first, then you can go”
  - Many different coordination mechanisms have been invented

- Goals of synchronization:
  - Liveness – ability to execute in a timely manner (informally, “something good eventually happens”)
  - Safety – avoid unintended interactions with shared data structures (informally, “nothing bad happens”)

Lock Synchronization

- Use a "Lock" to grant access to a *critical section* so that only one thread can operate there at a time
  - Executed in an uninterruptible (*i.e.* atomic) manner

- Lock Acquire
  - "Wait" until the lock is free, then take it

- Lock Release
  - Release the lock
  - If other threads are waiting, wake exactly one up to pass lock to

- Pseudocode:

```
// non-critical code

lock.acquire();    block
                   if locked
// critical section
lock.release();

// non-critical code
```

Lock API

- Locks are constructs that are provided by the operating system to help ensure synchronization
  - There are many types of locks (e.g. Mutex Lock, Spin Lock…)

- Only one thread can acquire a lock at a time,
  No thread can acquire that lock until it has been released

Milk Example – What is the Critical Section?

- What if we use a lock on the refrigerator?
  - Probably overkill – what if roommate wanted to get eggs?
- For performance reasons, only put what is necessary in the critical section
  - Lock *all* steps that must run uninterrupted; only lock the milk.
  - (*i.e.* must run as an atomic unit)

```
fridge.lock()
if (!milk) {
  buy milk
}
fridge.unlock()
```

↓

```
milk_lock.lock()
if (!milk) {
  buy milk
}
milk_lock.unlock()
```

pthread mutex locks

```
int pthread_mutex_init(pthread_mutex_t* mutex,
                       const pthread_mutexattr_t* attr);
```

- initializes the mutex object pointed to by mutex according to the mutex attributes specified in mutexattr.
  - You could even make locks shareable across processes…

```
int pthread_mutex_lock(pthread_mutex_t* mutex);
```

- If the mutex is currently unlocked, it becomes locked and owned by the calling thread.
- If the mutex is already locked by another thread, **pthread_mutex_lock**() ***suspends*** the calling thread until the mutex is unlocked.

```
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

- unlocks the given mutex. The mutex is ***assumed*** to be locked and owned by the calling thread on entrance to **pthread_mutex_unlock**(). Linux allows any thread to unlock a mutex, even if it isn't its owner. But, this isn't true across OS's.

```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```

Check the man page..

pthread Mutex Examples

- See `total.c`
  - Data race between threads
- See `total_locking.c`
  - Adding a mutex fixes our data race
- How does `total_locking` compare to sequential code and to `total`?
  - Likely *slower* than both– only 1 thread can increment at a time, and must deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
    - See `total_locking_better.c`
- How about with optimizations?
  - Let's see `total_locking_opt.c` with compiler optimizations.

Lecture Outline

- Threads Refresher
- Synchronization Mechanisms
- **Data Race vs Race Condition**
- Is a mutex needed? (Peterson's)
- It only gets worse

Poll Everywhere

Pause and think

```c
pthread_mutex_t lock;
bool print_ok = false;

void* thrd_fn1(void* arg) {
  pthread_mutex_lock(&lock);
  print_ok = true;
  pthread_mutex_unlock(&lock);
  return NULL;
}

void* thrd_fn2(void* arg) {
  pthread_mutex_lock(&lock);
  if (print_ok) {
    printf("print ok is true\n");
  } else {
    printf("print ok is false\n");
  }
  pthread_mutex_unlock(&lock);
  return NULL;
}

int main() {
  // assume main sets ups the threads & locks, etc.
```

- Does this code have a data race?
  - Can this program enter an “invalid” (unexpected or error) state from having concurrent memory accesses?
- Follow up: Does this code feel good?

Race Condition vs Data Race

- Data-Race: when there are concurrent accesses to a shared resource, with at least one write, that can cause the shared resource to enter an invalid or “unexpected” state.

- Race-Condition: Where the program has different behavior depending on the ordering of concurrent threads. This can happen even if all accesses to shared resources are “atomic” or “locked”
  - THINK SCHEDULER! SCHEDULER SCHEDULER SCHEDULER!

- The previous example has no data-race, but it does have a race condition
- Data-races are a subset of race-conditions

Thread Communication

- Sometimes threads may need to communicate with each other to know when they can perform operations

- Example: Producer and consumer threads
  - One thread creates tasks/data
  - One thread consumes the produced tasks/data to perform some operation
  - The consumer thread can only produce things once the creator has created them

- Need to make sure this communication has no data race or race condition

Lecture Outline

- Threads Refresher
- Synchronization Mechanisms
- Data Race vs Race Condition
- **Is a mutex needed? (Peterson's)**
- It only gets worse

Poll Everywhere

Pause and think

- Lets try a more complicated software approach..
- We create two threads running thread_code, one with `me = 0`, other thread has `me = 1`
- Each thread tries to increment sum_total. Does this work?

```
int sum_total = 0;
atomic bool flag[2] = {false, false};
atomic int turn = 0
void thread_code(void *arg) {
  int me = (int)arg;

  flag[me] = true;
  turn = 1 - me;
  while((flag[1-me] == true) && (turn != me)) { }
  ++sum_total;
  flag[me] = false;
}
```

Note: atomic <type> is a real thing just not as I've written it here.

Peterson’s Algorithm

- What we just did was Peterson's algorithm
- Why does it work? (using an analogy)
  - Each thread first declares that they want to enter the critical section by setting their flag
  - Each thread then states (once) that the other should “go first”.
    - This is done by setting the turn variable to 1 – me
    - One of these assignments to the turn variable will happen last, that is the one that decides who goes first
  - One of the thread goes first (decided by the value of turn) and accesses the critical section, before saying it is done (by changing their flag to false)

Peterson's Algorithm

- What we just did was Peterson's algorithm
- Why does it work?
  - Case1:
    If P0 enters critical section, flag[0] = true, turn = 0. It enters the critical section successfully.
  - Case2:
    If P0 and P1 enter critical section, flag[0] and flag[1] = true

    Race condition on turn. Suppose P0 sets turn = 0 first. Final value is turn = 1. P0 will get to run first.

Explanation

TIME

Thread 0

```
flag[0] = true
```

Thread 1

```
flag[1] = true
```

RACE

Thread 0:

```
turn = 1
turn = 1
```

Thread 1:

```
turn = 0
```

turn = ?

// suppose `turn = 1` **came after** `turn = 0`
// the `turn` **variable is set to 1**

turn = 1

Thread 0:

```
while(flag[1] == true
        && turn != 0)
```

Thread 1:

```
while(flag[0] == true
        && turn != 1)
```

```
++sum_total
```

```
flag[1] = false
```

Thread 0:

```
++sum_total
```

Peterson’s Assumptions

- Some operations are atomic:
  - Reading from the flag and turn variables cannot be interrupted
  - Writing to the flag and turn variables cannot be interrupted
  - E.g setting turn = 1 or 0 will set turn to 0 or 1, you can be interrupted before or after, but not “during” when turn may have some intermediate value that is not 0 or 1
- That the instructions are executed in the specific order laid out in the code

Atomicity

- Atomicity: An operation or set of operations on some data are *atomic* if the operation(s) are indivisible, that no other operation(s) on that same data can interrupt/interfere.

- Aside on terminology:
  - Often interchangeable with the term "Linearizability"
  - Atomic has a different (but similar-ish) meaning in the context of data bases and ACID.

Lecture Outline

- Threads Refresher
- Synchronization Mechanisms
- Data Race vs Race Condition
- Is a mutex needed? (Peterson's)
- **It only gets worse**

Poll Everywhere

Pause and think

- Can the assert fail here?

```
bool flag = false;
int x = 0;

void* thrd_fn1(void* arg) {
  x = 5;
  flag = true;
  return NULL;
}

void* thrd_fn2(void* arg) {
  if (flag) {
    assert(x == 5);  // crashes if x does not equal 5
  }
  return NULL;
}

int main() {
  // assume main creates two threads, one to run thrd_fn1 and another for thrd_fn2
```

Instruction & Memory Ordering

- Do we know that x is set before g is set?

```
bool g = false;
int x = 0

void some_func(int arg) {
  x = 5;
  g = true;
}
```

Instruction & Memory Ordering

- Do we know that x is set before g is set?

```
bool g = false;
int x = 0

void some_func(int arg) {
  x = 5;
  g = true;
}
```

NO

The compiler may generate instructions that sets g first and then t
The Processor may execute these out of order or at the same time

Why? Optimizations on program performance

Aside: Instruction & Memory Ordering

- The compiler may generate instructions with different ordering if it does not appear that it will affect the semantics of the function
  - Since `g = true;` is not affected by `x = 5;` then either one could execute first.
- The Processor may also execute these in a different order than what the compiler says
- Why? Optimizations on program performance
  - If you want to know more, look into "Out-of-Order Execution" and "Memory Order"

Aside: Memory Barriers

- How do we fix this?

- We can emit special instructions to the CPU and/or compiler to create a "memory barrier"
  - "all memory accesses before the barrier are guaranteed to happen before the memory accesses that come after the barrier"
  - A way to enforce an order in which memory accesses are ordered by the compiler and the CPU

- The further you go the discrepancy becomes more nuanced.
  - Do we want to force mem access to occur in the order of the written program?
  - Do we want to force that all memory access/modifications *to complete* before the next instruction executes?

Lecture Outline

- Threads Refresher
- Synchronization Mechanisms
- Data Race vs Race Condition
- Is a mutex needed? (Peterson's)
- It only gets worse

Why Threads?

- Advantages:
  - You (mostly) write sequential-looking code
  - Threads can run in parallel if you have multiple CPUs/cores

- Disadvantages:
  - If threads share data, you need locks or other synchronization
    - Very bug-prone and difficult to debug
  - Threads can introduce overhead
    - Lock contention, context switch overhead, and other issues
  - Need language support for threads

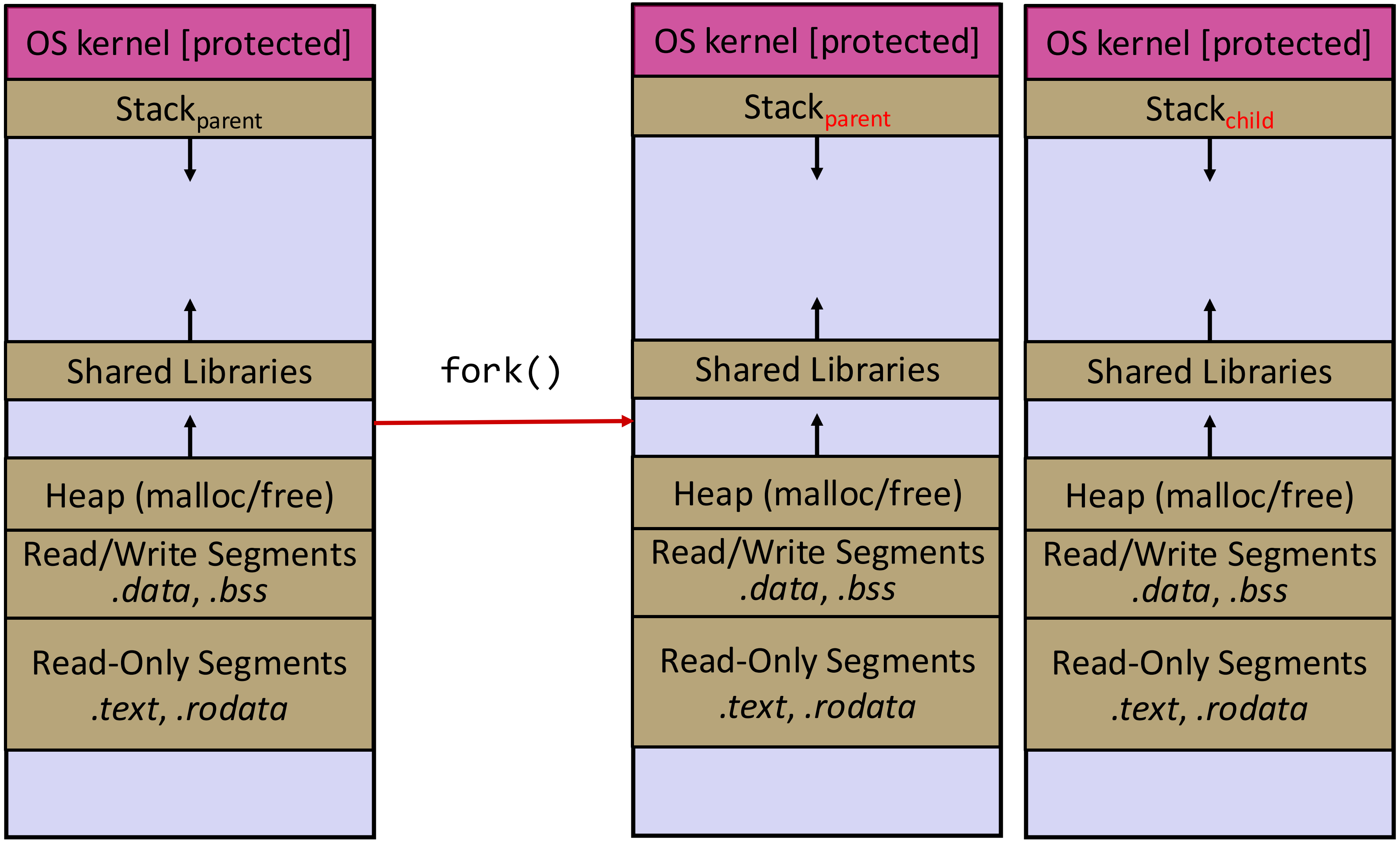
That's all!

- See you next time!

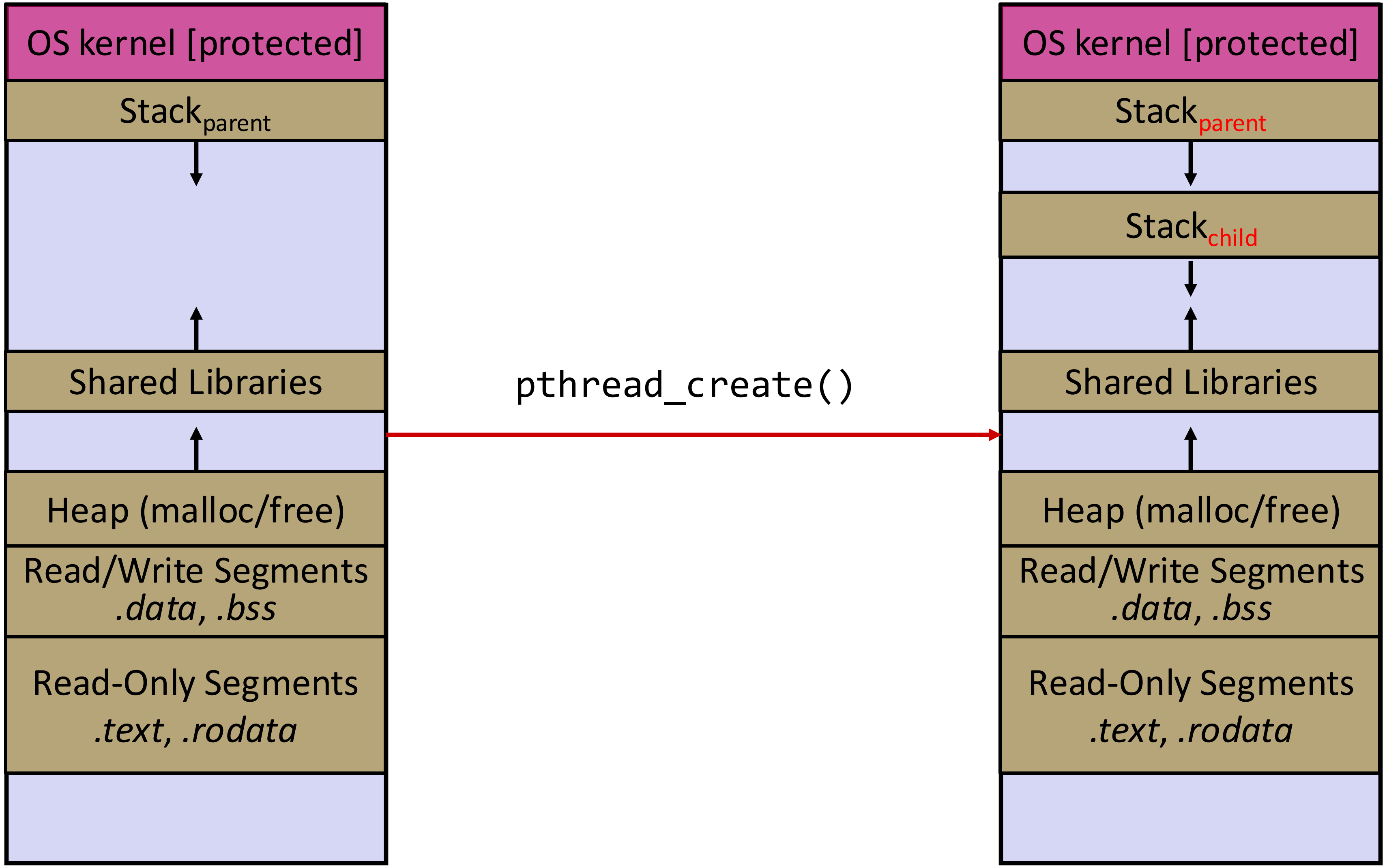
Threads vs. Processes

- In most modern OS's:
  - A Process has a unique: address space, OS resources, & security attributes
  - A Thread has a unique: stack, stack pointer, program counter, & registers
  - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

Threads vs. Processes



Threads vs. Processes



Alternative: Processes

- What if we forked processes instead of threads?

- Advantages:
  - No shared memory between processes
  - No need for language support; OS provides "fork"
  - Processes are isolated. If one crashes, other processes keep going

- Disadvantages:
  - More overhead than threads during creation and context switching (Context switching == switching between threads/processes)
  - Cannot easily share memory between processes – typically communicate through the file system